Blockchain Assessment 2

Introduction

The primary objective of this program is to implement and build upon the solutions previously created; the focus now will be to build a robust, simulated interactive program. This program will demonstrate how Blockchain and Cryptography concepts can be implemented to distribute records across four different warehouse locations and retrieve them securely. Using a traditional approach allows room for lots of security and data vulnerabilities, such as data tampering, unauthorized access to databases. Distributed Ledger Technology (DLT) allows access to only authorized users, ensures data integrity, and provides a way to oversee data changes and who made them Nevil, S. (2024). Implementation will include key Blockchain concepts RSA-based digital signatures, consensus protocol, and Harn's identity-based multi-signature scheme.

Objectives

This Program will be created with two parts, with two different objectives. The implementation must consist of a user interface. This will be built with Flask and HTML. Cryptographic operations will be done in back backend Python.

- RSA Digital Signature Implementation. Updating and adding new records with consensus.
- The second program will focus on securely retrieving records stored within the inventory

There are four main objectives of functionality which will be implemented
within the program:
New Record Submission:

When a record is entered in a node, it must be verified for its authenticity and data integrity, and consensus must be formed before permanently being stored on the blockchain.

- When a user enters a record into the DLT system, the system must ensure it is entered from a legitimate inventory node.
- Malicious actors do not tamper with the record during transmission; data integrity must be ensured.
- Only after consensus is formed within all nodes, the record is permanently stored within the block chain. blockchain

Multi-Signature-Based Query Verification:

When a Procurement Officer queries the DLT system for an item, all inventory nodes must jointly verify the queried record using the Harn identity-based multi-signature scheme.

- The data received by the Procurement Officer must be accurate, untampered and collectively approved by all Nodes.
- The system must prevent faulty nodes from providing inconsistent or forged results.

Data Encryption Before Response Transmission:

The queried record, once approved by all inventories, must be encrypted before being sent to the authorized user.

- The record is encrypted to ensure only the query user can decrypt and access the record.

Scope

- Simulation of a DLT inventory system using separate text databases
- RSA cryptography used to sign and create signatures to validate record integrity and authenticity.
- Record will be sent to each node and verification will happen at each node.

What's assumed or left out?

- As this demonstration is a simulation of real world networking setup e.g Servers, Routers etc, will not be done.
- Databases will be simulated using text files for storage.
- Merkle tree, Merkle tree root calculations are excluded.

- Consensus mechanism is simulated at each node.
- Multi-Signature-Based Query Verification and RSA encryption for retrieving records.
- Back end will include all the logic needed.
- Front end will used to interact with the simulated nodes.
- Use of external cryptography libraries to sign encrypt etc.

Implementation of Digital Signatures and Consensus Protocol
Digital Signature Implementation

Tools and Libraries used for implementation.
Backend:

- Vs Code used as Integrated development environment
- Programming language used Python 3.11.9
- Flask used to create web server and handle user inputs
- Libraries
  >hashilb
  >math
  >random

Front End

- Language: HTML,CSS
- Templating engine Jinja(part of flask to generate HTML code)

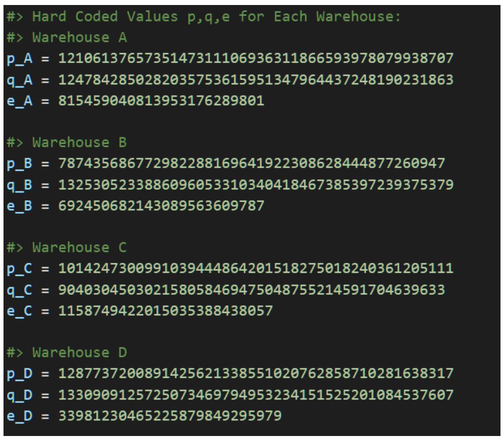
Process Approach when creating this program:

The logic for all the code below follows the following steps listed to achieve record submission:

- Step 1: RSA algorithm used to generate public and private keys (pk,sk)
- Step 2:pk secret key is used to sign the record from an inventory.
- Step 3:Signed record is sent to all Nodes and all nodes verify the record with public key.
- Step 4:Only if verification is approved, the record is added into a temp pool.
- Step 5:Only after consensus is formed the record is stored in all the inventories.

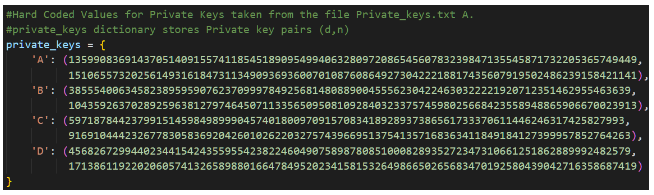
Screenshots of Key code snippets found below:

Backend



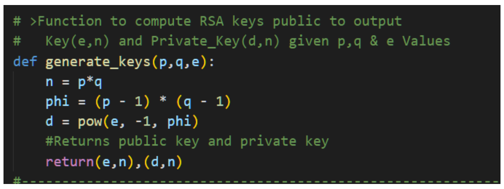
```
#> Hard Coded Values p,q,e for Each Warehouse:
#> Warehouse A
p_A = 1210613765735147311106936311866593978079938707
q_A = 1247842850282035753615951347964437248190231863
e_A = 815459040813953176289801

#> Warehouse B
p_B = 787435686772982288169641922308628444877260947
q_B = 1325305233886096053310340418467385397239375379
e_B = 692450682143089563609787

#> Warehouse C
p_C = 1014247300991039444864201518275018240361205111
q_C = 904030450302158058469475048755214591704639633
e_C = 1158749422015035388438057

#> Warehouse D
p_D = 1287737200891425621338551020762858710281638317
q_D = 1330909125725073469794953234151525201084537607
e_D = 33981230465225879849295979
```

Hard coded PQE values for each warehouse, these values were called in the function generate keys to output pk and sk keys in two different files.



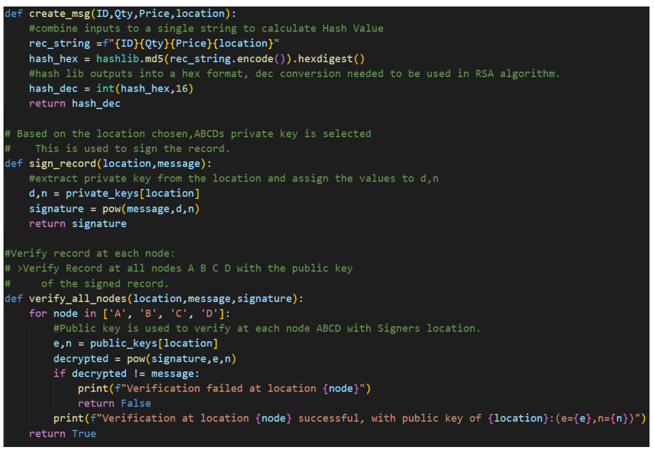
```
#Hard Coded Values for Private Keys taken from the file Private_keys.txt A.
#private_keys dictionary stores Private key pairs (d,n)
private_keys = {
    'A': (1359908369143705140915574118545189095499406328097208654560783239847135545871732205365749449,
          1510655732025614931618473113490936936007010876086492730422218817435607919502486239158421141),
    'B': (385554006345823895959076237099978492568148088900455562304224630322221920712351462955463639,
          1043592637028925963812797464507113356509508109284032337574598025668423558948865906670023913),
    'C': (59718784423799151459849899904574018009709157083418928937386561733370611446246317425827993,
          916910444232677830583692042601026220327574396695137541357168363411849184127399957852764263),
    'D': (456826729944023441542435595542382246049075898780851000828935272347310661251862889992482579,
          1713861192202060574132658988016647849520234158153264986650265683470192580439042716358687419)
}
```

Dictionary for public and private keys created so these values can be easily called onto sign and verify.



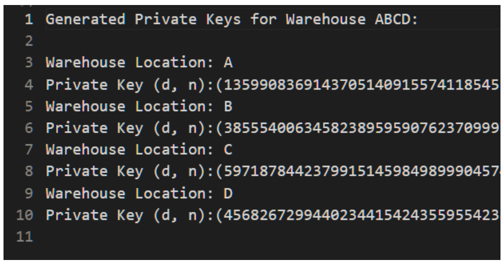
```
# >Function to compute RSA keys public to output
#   Key(e,n) and Private_Key(d,n) given p,q & e Values
def generate_keys(p,q,e):
    n = p*q
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)
    #Returns public key and private key
    return(e,n),(d,n)
#------------------------------------------------------------
```

Function defined to generate the RSA keys takes in pqe values to out put keys .



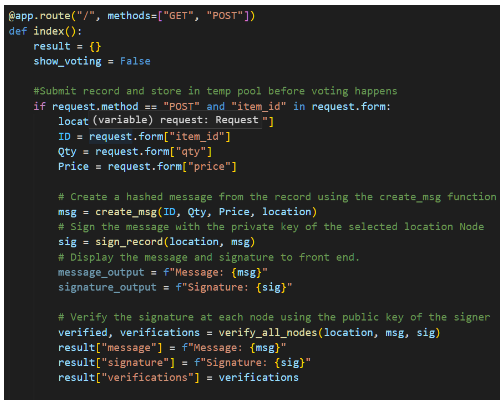
```
def create_msg(ID,Qty,Price,location):
    #combine inputs to a single string to calculate Hash Value
    rec_string =f"{ID}{Qty}{Price}{location}"
    hash_hex = hashlib.md5(rec_string.encode()).hexdigest()
    #hash lib outputs into a hex format, dec conversion needed to be used in RSA algorithm.
    hash_dec = int(hash_hex,16)
    return hash_dec

# Based on the location chosen,ABCDs private key is selected
#    This is used to sign the record.
def sign_record(location,message):
    #extract private key from the location and assign the values to d,n
    d,n = private_keys[location]
    signature = pow(message,d,n)
    return signature

#Verify record at each node:
# >Verify Record at all nodes A B C D with the public key
#     of the signed record.
def verify_all_nodes(location,message,signature):
    for node in ['A', 'B', 'C', 'D']:
        #Public key is used to verify at each node ABCD with Signers location.
        e,n = public_keys[location]
        decrypted = pow(signature,e,n)
        if decrypted != message:
            print(f"Verification failed at location {node}")
            return False
        print(f"Verification at location {node} successful, with public key of {location}:(e={e},n={n})")
    return True
```

function defined to create hash msg from user input which will then be signed in def sign _record function. Next step verify all nodes will take in this hashed message, use location to select keys and verify all records.



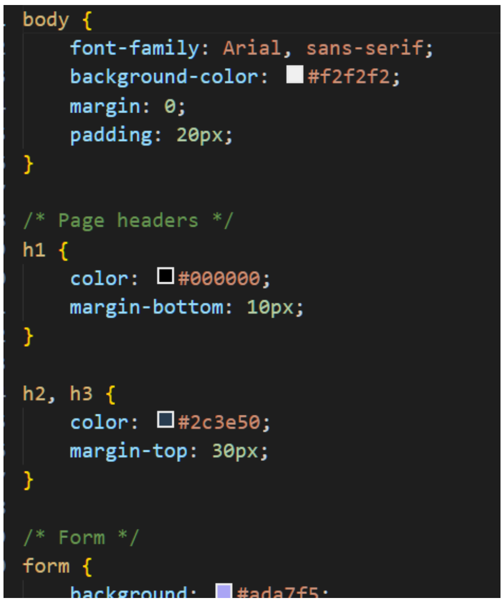
```
Generated Private Keys for Warehouse ABCD:

Warehouse Location: A
Private Key (d, n):(135990836914370514091557411854
Warehouse Location: B
Private Key (d, n):(385554006345823895959076237099
Warehouse Location: C
Private Key (d, n):(597187844237991514598498999045
Warehouse Location: D
Private Key (d, n):(456826729944023441542435595542
```

Keys stored in two separate files.



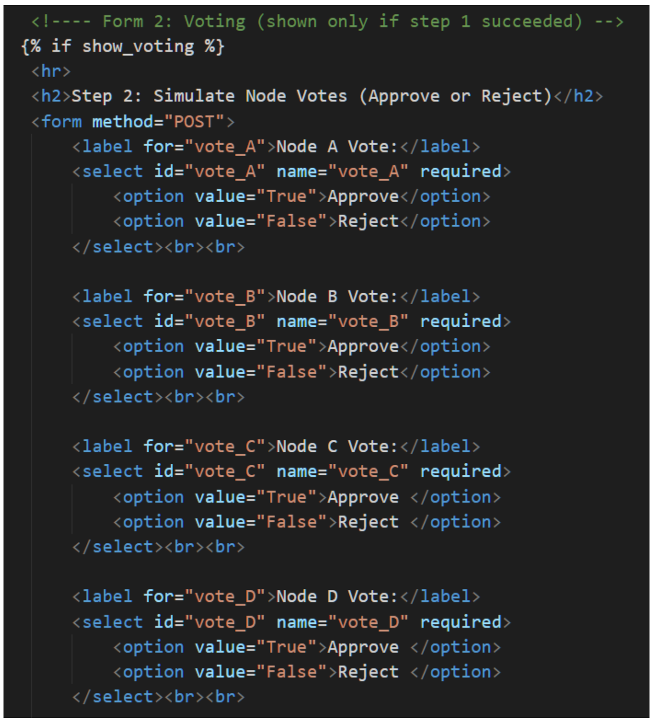
```python
@app.route("/", methods=["GET", "POST"])
def index():
    result = {}
    show_voting = False

    #Submit record and store in temp pool before voting happens
    if request.method == "POST" and "item_id" in request.form:
        locat (variable) request: Request "]
        ID = request.form["item_id"]
        Qty = request.form["qty"]
        Price = request.form["price"]

        # Create a hashed message from the record using the create_msg function
        msg = create_msg(ID, Qty, Price, location)
        # Sign the message with the private key of the selected location Node
        sig = sign_record(location, msg)
        # Display the message and signature to front end.
        message_output = f"Message: {msg}"
        signature_output = f"Signature: {sig}"

        # Verify the signature at each node using the public key of the signer
        verified, verifications = verify_all_nodes(location, msg, sig)
        result["message"] = f"Message: {msg}"
        result["signature"] = f"Signature: {sig}"
        result["verifications"] = verifications
```

@app is a decorator for web framework it tells flask what url should trigger which function inside the program.



```css
body {
    font-family: Arial, sans-serif;
    background-color: #f2f2f2;
    margin: 0;
    padding: 20px;
}

/* Page headers */
h1 {
    color: #000000;
    margin-bottom: 10px;
}

h2, h3 {
    color: #2c3e50;
    margin-top: 30px;
}

/* Form */
form {
    background: #ada7f5;
```

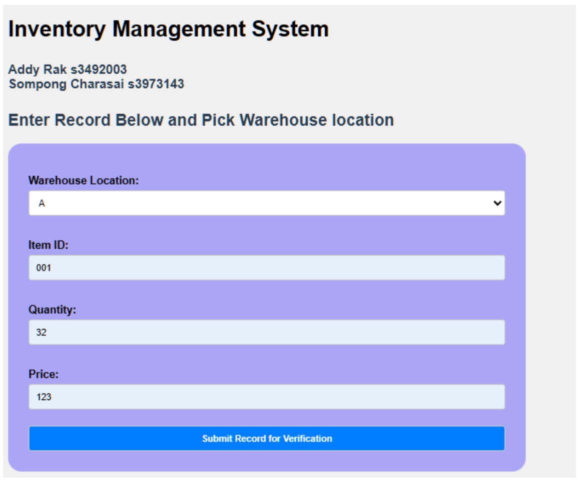
css file used to decorate html page



```html
<!---- Form 2: Voting (shown only if step 1 succeeded) -->
{% if show_voting %}
<hr>
<h2>Step 2: Simulate Node Votes (Approve or Reject)</h2>
<form method="POST">
    <label for="vote_A">Node A Vote:</label>
    <select id="vote_A" name="vote_A" required>
        <option value="True">Approve</option>
        <option value="False">Reject</option>
    </select><br><br>

    <label for="vote_B">Node B Vote:</label>
    <select id="vote_B" name="vote_B" required>
        <option value="True">Approve</option>
        <option value="False">Reject</option>
    </select><br><br>

    <label for="vote_C">Node C Vote:</label>
    <select id="vote_C" name="vote_C" required>
        <option value="True">Approve </option>
        <option value="False">Reject </option>
    </select><br><br>

    <label for="vote_D">Node D Vote:</label>
    <select id="vote_D" name="vote_D" required>
        <option value="True">Approve </option>
        <option value="False">Reject </option>
    </select><br><br>
```

HTML page designs the page which will take user input.

Front End


Inventory Management System

Addy Rak s3492003
Sompong Charasai s3973143

Enter Record Below and Pick Warehouse location

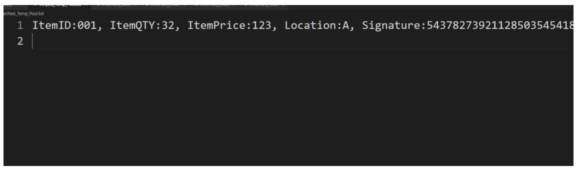
Warehouse Location: A

Item ID: 001

Quantity: 32

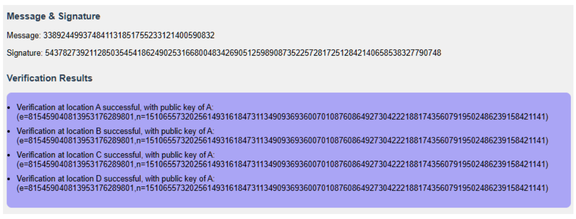
Price: 123

User enters record and record is stored in temp pool after submission



```
ItemID:001, ItemQTY:32, ItemPrice:123, Location:A, Signature:54378273921128503545418
```

Temp pool text file

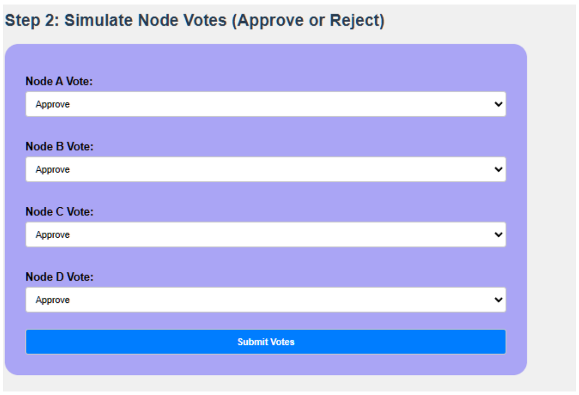

Message & Signature

Message: 338924499374841131851755233121400590832

Signature: 543782739211285035454186249025316680048342690512598908735225728172512842140658538327790748

Verification Results

- Verification at location A successful, with public key of A: (e=815459040813953176289801,n=1510655732025614931618473113490936936007010876086492730422218817435607919502486239158421141)
- Verification at location B successful, with public key of A: (e=815459040813953176289801,n=1510655732025614931618473113490936936007010876086492730422218817435607919502486239158421141)
- Verification at location C successful, with public key of A: (e=815459040813953176289801,n=1510655732025614931618473113490936936007010876086492730422218817435607919502486239158421141)
- Verification at location D successful, with public key of A: (e=815459040813953176289801,n=1510655732025614931618473113490936936007010876086492730422218817435607919502486239158421141)


Step 2: Simulate Node Votes (Approve or Reject)

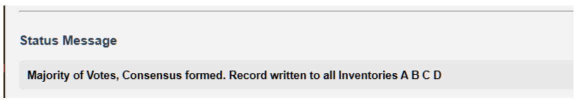
Node A Vote: Approve

Node B Vote: Approve

Node C Vote: Approve

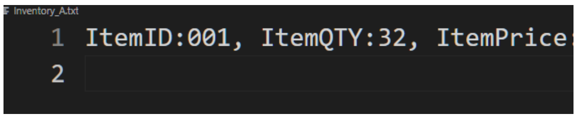
Node D Vote: Approve

Only after majority of nodes authenticate the record 3/4 the record gets added to the main pool

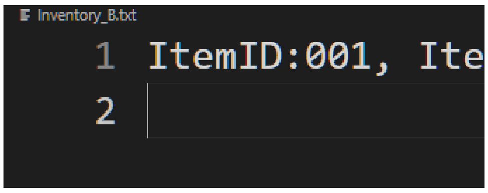

Status Message

Majority of Votes, Consensus formed. Record written to all Inventories A B C D

All inventories simultaneously update the record after consensus is formed!

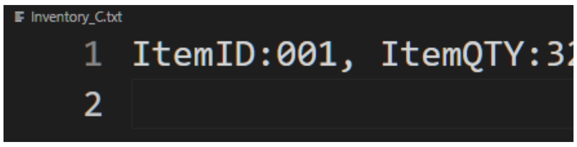


```
ItemID:001, ItemQTY:32, ItemPrice
```



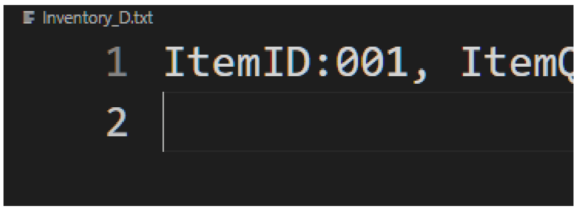
Inventory_B.txt

```
ItemID:001, Ite
```



Inventory_C.txt

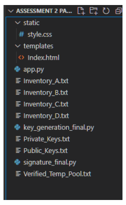
```
ItemID:001, ItemQTY:3
```



Inventory_D.txt

```
ItemID:001, ItemQ
```

Data Structure



```
ASSESSMENT 2 PA...
  static
    style.css
  templates
    Index.html
  app.py
  Inventory_A.txt
  Inventory_B.txt
  Inventory_C.txt
  Inventory_D.txt
  key_generation_final.py
  Private_Keys.txt
  Public_Keys.txt
  signature_final.py
  Verified_Temp_Pool.txt
```

This is how the files are structured in the project

Consensus Protocol Justification

Inventory management systems are ideal for Proof of Authority (PoA) as a consensus protocol because the participants are known and limited. PoA allows the authority nodes (validators) to verify and create blocks. It allows inventory management to operate with low computational overhead, high-speed processing, and high throughput because inventory is updated so frequently. To simulate this in the program verification validation at each node is simulated by creating a voting system. Only if all the authority nodes agree; the record is stored in the main Inventory nodes ABCD.

PoA provides a model for levels of authority where appropriate permission can be enabled. PoA, however, inappropriately centralizes decision-making authority and operates under the possibility of collusion within a small number of validators that lack adequate decision-making.

Since fewer validation nodes exist, there is an accepted risk of lower fault tolerance. By contrast, Proof of Work (PoW) decision-making for inventory management systems is hampered by high computational resource usage and verification processing delays, and Proof of Stake (PoS) adds levels of complexity associated with token economics and the challenges of wealth concentration, creating concerns around fairness. Therefore, in cooperation where participants are known, like inventory management systems, the appropriate consensus to select would be PoA.

Multi-Signature Query Verification & Secure Delivery Implementation

Tools and Libraries used for implementation.

Backend:

- Vs Code used as Integrated development environment
- Programming language used Python 3.11.9
- Flask used to create web server and handle user inputs
- Libraries
    - hashlib
    - math
    - random

Front End

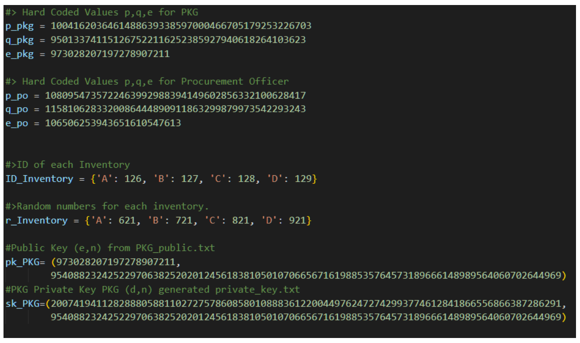
- Language: HTML,CSS
- Templating engine Jinja(part of flask to generate HTML code)

Overview of the Process Approach how program works:

- Step 1:Front end Procurement Officer submits a query by entering an ID in the form. This form is created in the <HTML> code snippets found below. CSS is used for some basic styling the User interface
- Step 2: Each inventory node searches its local database (text file for each database) for the retrieval process.

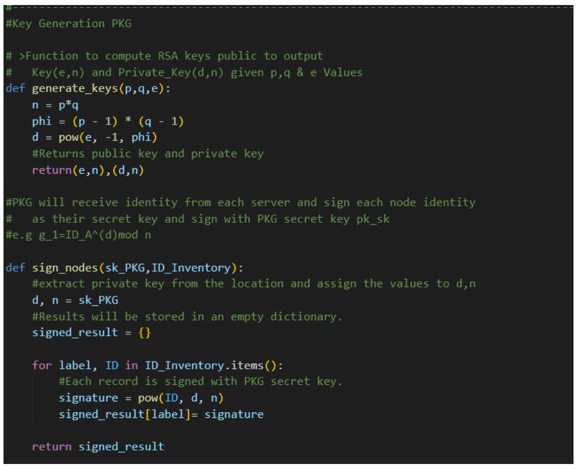
- Step 3: Each node generates partial signature using the Harn identity-based multi-signature algorithm
- Step 4: PKG combines all the partial signatures of each node to combine into one signature.
- Step 5: Consensus check is done by all nodes agree with the aggregated signature.
- Step 6: Next the verified record is encrypted using the Procurement Officer's public RSA key
- Step 7:Procurement Officer private key is finally used to decrypt and read the private record. Only Procurement Officer can see this record using private key known only to him/her.

Backend



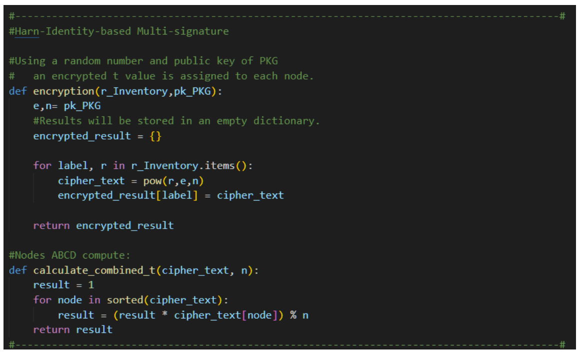
```
#> Hard Coded Values p,q,e for PKG
p_pkg = 1004162036461488639338597000466705179253226703
q_pkg = 950133741151267522116252385927940618264103623
e_pkg = 973028207197278907211

#> Hard Coded Values p,q,e for Procurement Officer
p_po = 1080954735722463992988394149602856332100628417
q_po = 1158106283320086444890911863299879973542293243
e_po = 106506253943651610547613


#>ID of each Inventory
ID_Inventory = {'A': 126, 'B': 127, 'C': 128, 'D': 129}

#>Random numbers for each inventory.
r_Inventory = {'A': 621, 'B': 721, 'C': 821, 'D': 921}

#Public Key (e,n) from PKG_public.txt
pk_PKG= (973028207197278907211,
        954088232425229706382520201245618381050107066567161988535764573189666148989564060702644969)
#PKG Private Key PKG (d,n) generated private_key.txt
sk_PKG=(200741941128288805881102727578608580108883612200449762472742993774612841866556866387286291,
        954088232425229706382520201245618381050107066567161988535764573189666148989564060702644969)
```

Hard coded values for PKG pqe , ID for each inventory, Random numbers. Generated public and private key.



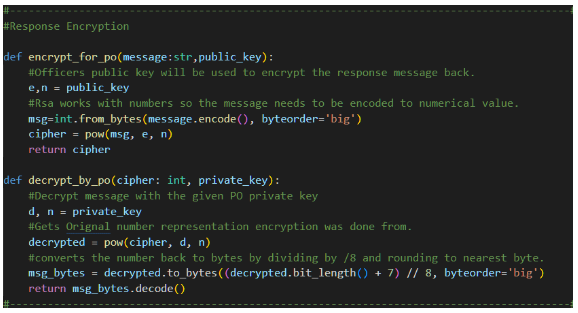
```
#------------------------------------------------------------------------------
#Key Generation PKG

# >Function to compute RSA keys public to output
#   Key(e,n) and Private_Key(d,n) given p,q & e Values
def generate_keys(p,q,e):
    n = p*q
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)
    #Returns public key and private key
    return(e,n),(d,n)

#PKG will receive identity from each server and sign each node identity
#   as their secret key and sign with PKG secret key pk_sk
#e.g g_1=ID_A^(d)mod n

def sign_nodes(sk_PKG,ID_Inventory):
    #extract private key from the location and assign the values to d,n
    d, n = sk_PKG
    #Results will be stored in an empty dictionary.
    signed_result = {}

    for label, ID in ID_Inventory.items():
        #Each record is signed with PKG secret key.
        signature = pow(ID, d, n)
        signed_result[label]= signature

    return signed_result
```

PKG key generation to create PKG pk and sk values.



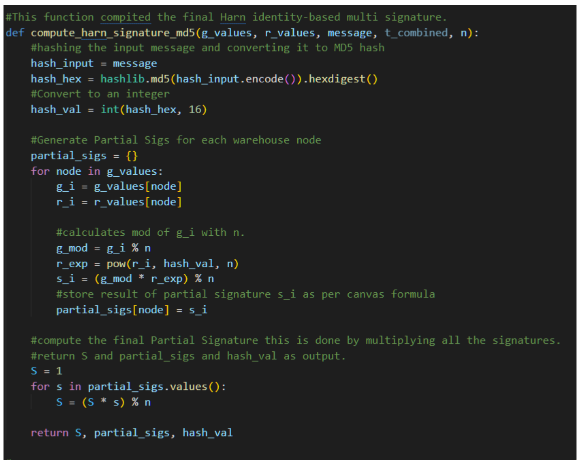
```
#----------------------------------------------------------------------------#
#Harn-Identity-based Multi-signature

#Using a random number and public key of PKG
#   an encrypted t value is assigned to each node.
def encryption(r_Inventory,pk_PKG):
    e,n= pk_PKG
    #Results will be stored in an empty dictionary.
    encrypted_result = {}

    for label, r in r_Inventory.items():
        cipher_text = pow(r,e,n)
        encrypted_result[label] = cipher_text

    return encrypted_result

#Nodes ABCD compute:
def calculate_combined_t(cipher_text, n):
    result = 1
    for node in sorted(cipher_text):
        result = (result * cipher_text[node]) % n
    return result
#----------------------------------------------------------------------------#
```

Def encryption used the dictionary for random number and pk private PKG key to encrypt the result.



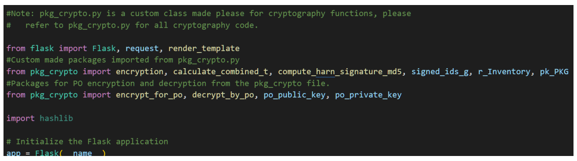
```
#------------------------------------------------------------------------------#
#Response Encryption

def encrypt_for_po(message:str,public_key):
    #Officers public key will be used to encrypt the response message back.
    e,n = public_key
    #Rsa works with numbers so the message needs to be encoded to numerical value.
    msg=int.from_bytes(message.encode(), byteorder='big')
    cipher = pow(msg, e, n)
    return cipher

def decrypt_by_po(cipher: int, private_key):
    #Decrypt message with the given PO private key
    d, n = private_key
    #Gets Orignal number representation encryption was done from.
    decrypted = pow(cipher, d, n)
    #converts the number back to bytes by dividing by /8 and rounding to nearest byte.
    msg_bytes = decrypted.to_bytes((decrypted.bit_length() + 7) // 8, byteorder='big')
    return msg_bytes.decode()
#------------------------------------------------------------------------------#
```

RSA encryption and decryption for PO officer.



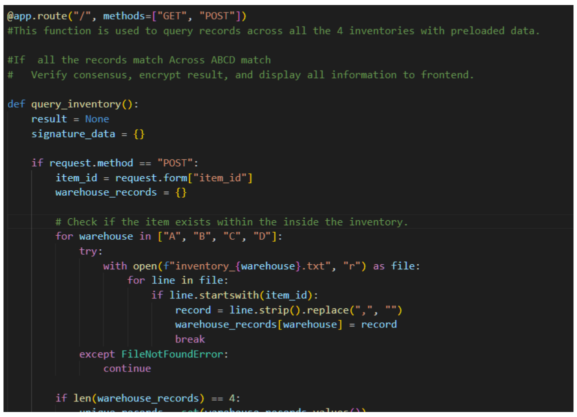
```python
#This function compited the final Harn identity-based multi signature.
def compute_harn_signature_md5(g_values, r_values, message, t_combined, n):
    #hashing the input message and converting it to MD5 hash
    hash_input = message
    hash_hex = hashlib.md5(hash_input.encode()).hexdigest()
    #Convert to an integer
    hash_val = int(hash_hex, 16)

    #Generate Partial Sigs for each warehouse node
    partial_sigs = {}
    for node in g_values:
        g_i = g_values[node]
        r_i = r_values[node]

        #calculates mod of g_i with n.
        g_mod = g_i % n
        r_exp = pow(r_i, hash_val, n)
        s_i = (g_mod * r_exp) % n
        #store result of partial signature s_i as per canvas formula
        partial_sigs[node] = s_i

    #compute the final Partial Signature this is done by multiplying all the signatures.
    #return S and partial_sigs and hash_val as output.
    S = 1
    for s in partial_sigs.values():
        S = (S * s) % n

    return S, partial_sigs, hash_val
```

Create hash input and MD5 Hash, convert to integer. This function is a key function that calculates Harn identity multi signature by generating partial signatures and compute final aggregated signature.

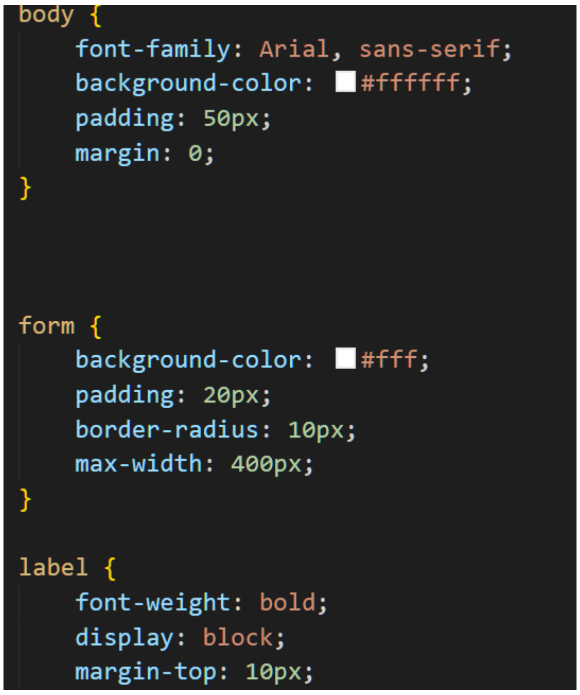


```python
#Note: pkg_crypto.py is a custom class made please for cryptography functions, please
#   refer to pkg_crypto.py for all cryptography code.

from flask import Flask, request, render_template
#Custom made packages imported from pkg_crypto.py
from pkg_crypto import encryption, calculate_combined_t, compute_harn_signature_md5, signed_ids_g, r_Inventory, pk_PKG
#Packages for PO encryption and decryption from the pkg_crypto file.
from pkg_crypto import encrypt_for_po, decrypt_by_po, po_public_key, po_private_key

import hashlib

# Initialize the Flask application
app = Flask(__name__)
```

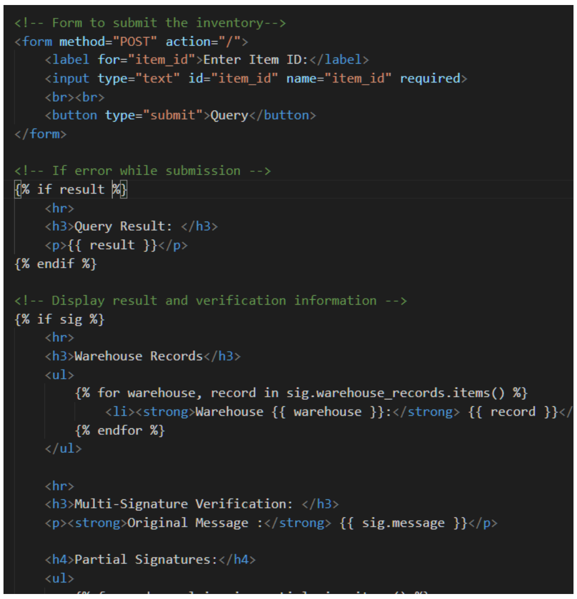
custom made crypto functions are imported no external crypto libraries are used.



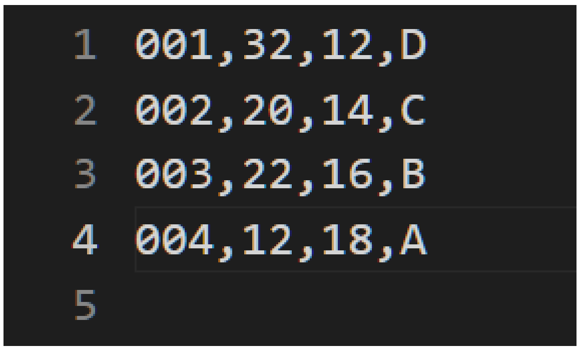
```python
@app.route("/", methods=["GET", "POST"])
#This function is used to query records across all the 4 inventories with preloaded data.

#If  all the records match Across ABCD match
#   Verify consensus, encrypt result, and display all information to frontend.

def query_inventory():
    result = None
    signature_data = {}

    if request.method == "POST":
        item_id = request.form["item_id"]
        warehouse_records = {}

        # Check if the item exists within the inside the inventory.
        for warehouse in ["A", "B", "C", "D"]:
            try:
                with open(f"inventory_{warehouse}.txt", "r") as file:
                    for line in file:
                        if line.startswith(item_id):
                            record = line.strip().replace(",", "")
                            warehouse_records[warehouse] = record
                            break
            except FileNotFoundError:
                continue

        if len(warehouse_records) == 4:
```



```css
body {
    font-family: Arial, sans-serif;
    background-color: #ffffff;
    padding: 50px;
    margin: 0;
}



form {
    background-color: #fff;
    padding: 20px;
    border-radius: 10px;
    max-width: 400px;
}

label {
    font-weight: bold;
    display: block;
    margin-top: 10px;
```



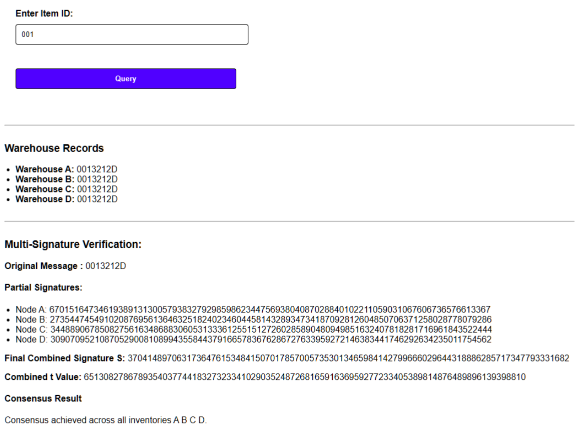
```html
<!-- Form to submit the inventory-->
<form method="POST" action="/">
    <label for="item_id">Enter Item ID:</label>
    <input type="text" id="item_id" name="item_id" required>
    <br><br>
    <button type="submit">Query</button>
</form>

<!-- If error while submission -->
{% if result %}
    <hr>
    <h3>Query Result: </h3>
    <p>{{ result }}</p>
{% endif %}

<!-- Display result and verification information -->
{% if sig %}
    <hr>
    <h3>Warehouse Records</h3>
    <ul>
        {% for warehouse, record in sig.warehouse_records.items() %}
            <li><strong>Warehouse {{ warehouse }}:</strong> {{ record }}</
        {% endfor %}
    </ul>

    <hr>
    <h3>Multi-Signature Verification: </h3>
    <p><strong>Original Message :</strong> {{ sig.message }}</p>

    <h4>Partial Signatures:</h4>
    <ul>
```



```
001,32,12,D
002,20,14,C
003,22,16,B
004,12,18,A
```

preloaded values in each inventory ABCD

Front End



Enter Item ID:

001

Warehouse Records

- **Warehouse A:** 0013212D
- **Warehouse B:** 0013212D
- **Warehouse C:** 0013212D
- **Warehouse D:** 0013212D

Multi-Signature Verification:

Original Message : 0013212D

Partial Signatures:

- Node A: 670151647346193891313005793832792985986234475693804087028840102211059031067606736576613367
- Node B: 273544745491020876956136463251824023460445814328934734187092812604850706371258028778079286
- Node C: 344889067850827561634868830605313336125515127260285890480949851632407818281716961843522444
- Node D: 309070952108705290081089943558443791665783676286727633959272146383441746292634235011754562

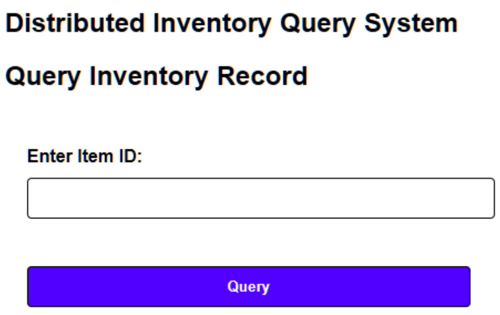
Final Combined Signature S: 370414897063173647615348415070178570057353013465984142799666029644318886285717347793331682

Combined t Value: 651308278678935403774418327323341029035248726816591636959277233405389814876489896139398810

Consensus Result

Consensus achieved across all inventories A B C D.

User is able to see the over view of all inventories, and signature verification.

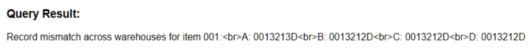


Distributed Inventory Query System

Query Inventory Record

Enter Item ID:

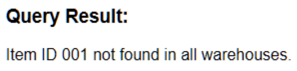
User Inputs the query.



Query Result:

Record mismatch across warehouses for item 001:
A: 0013213D
B: 0013212D
C: 0013212D
D: 0013212D

If the record is missing, there is a mis match notification.

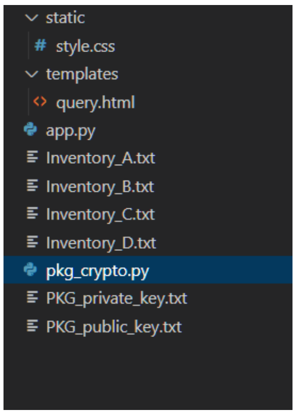


Query Result:

Item ID 001 not found in all warehouses.

If the query does not exist the user will get notified in the front end:

Data Structure



```
∨ static
    # style.css
∨ templates
    <> query.html
  app.py
  Inventory_A.txt
  Inventory_B.txt
  Inventory_C.txt
  Inventory_D.txt
  pkg_crypto.py
  PKG_private_key.txt
  PKG_public_key.txt
```

Data structure used for the program.

Conclusion

Using fundamental blockchain concepts, the implementation effectively replicates a safe and decentralised inventory management system. Strong multi-signature verification using Harn's identity-based scheme, consensus-based record validation via Proof-of-Authority (PoA), encrypted data response transmission, and secure New Record Submission with RSA digital signatures are some of the key features.

When utilising this system, users can anticipate

Accurate inventory data verification and authentication.
Data integrity is ensured by tamper-proof consensus mechanisms.
Data leaks are prevented by secure, encrypted communication.
Multi-party approval creates a sense of collective trust.

References

ErbaAitbayev. (2022). Simple Python RSA for digital signature with hashing implementation. For hashing SHA-256 from hashlib library is used [Source code]. GitHub.

https://gist.github.com/ErbaAitbayev/8f491c04af5fc1874e2b0744965a732b
GeeksforGeeks. (2021). RSA digital signature scheme using Python. Retrieved May 23, 2025, from
https://www.geeksforgeeks.org/rsa-digital-signaturescheme-using-python/
GeeksforGeeks. (n.d.). Programming tutorials and examples. Retrieved May 23, 2025, from
https://www.geeksforgeeks.org/
Grinberg, M. (n.d.). The Flask mega-tutorial part I: Hello, world. Retrieved May 23, 2025, from
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorialpart-i-hello-world
Helpjuice. (n.d.). Technical documentation: Purpose and best practices. Retrieved May 23, 2025, from
https://helpjuice.com/blog/technicaldocumentation
Pallets Projects. (n.d.). Flask tutorial: Layout. Flask Documentation (v2.0). Retrieved May 23, 2025, from

https://flask.palletsprojects.com/en/stable/tutorial/layout/
Python Software Foundation. (n.d.). hashlib — Secure hashes and message digests. Python 3.11. Retrieved May 23, 2025, from

https://docs.python.org/3/library/hashlib.html
Unsplash. (n.d.). A 3D image of a cube made of cubes [Photograph]. Retrieved May 23, 2025, from
https://unsplash.com/photos/a-3d-image-of-a-cubemade-of-cubes-UxDU0Gg5pqQ
W3Schools. (n.d.). HTML and CSS tutorials. Retrieved May 23, 2025, from

https://www.w3schools.com/